Mission Director Guide

Please note that this is officially-maintained documentation.

To ensure that you can rely on the information having been checked by Egosoft, you will not be able to edit this page.

The Mission Director (MD) is a subsystem of the game and interprets mission scripts, which are written in an XML-based language. The Mission Director in X Rebirth and X4 is based on the MD in X3: Terran Conflict, with some major changes based on feedback from MD users.

An introduction to the original MD can be found in the Egosoft forums. There is also a PDF guide for the X3 Mission Director, which is partially used as a template for this document.

This document is primarily supposed to be a guide for MD users (people who use the MD to develop missions or write other MD scripts), not for MD programmers (people who work on the MD engine in C++).

The general MD scripting system is the same in XR and X4, so this guide applies to both games. However, each game has its own set of supported script features (i.e. actions, conditions and properties), so in general scripts from different games are not compatible.

Table of Contents

- MD scripts
  - Script debug output
- MD script structure
  - Cues
  - Conditions
  - Actions
- Libraries
  - Library Parameters
- Instantiation
  - Cleaning up instances explicitly
  - Access to instances
  - Pitfalls
- Expressions
  - Numeric data types and suffixes
  - Operators
    - Operator precedence rules
    - Type conversion
    - Boolean operators
  - Strings and formatting
  - Lists
  - Tables
  - Value properties
    - Lookup tests and suppressing errors
    - Static lookups
    - Player properties
    - Safe properties
    - Money and time formatting
    - Complete property documentation
- MD refreshing and patching
  - Details and restrictions
  - Patching
- Common attribute groups
  - Value comparisons
  - Random ranges
- Variables and namespaces
  - Creating and removing variables
  - Accessing remote variables
  - Namespaces
    - Defining a cue's namespace

MD scripts

MD scripts are not necessarily missions. An MD file can contain a part of a mission, multiple missions, or no mission at all, as the MD is used for more than just missions.

MD files are XML files located in the game folder `md`. All XML files in that folder are loaded at game start. The file names are irrelevant, since the internally used script names are read from the XML root nodes. However, it's recommended to keep file name and internal script name identical to avoid having to look up the names.

To edit MD scripts, an XML editing tool is needed. Microsoft Visual Studio (if available) or Microsoft Visual Web Developer (for free) are highly recommended because they have pretty good support for XML schemas (XSD). The provided Mission Director schema files help you create the XML file by displaying all available tags and attributes as you edit the XML.

This functionality is only available if the schema files **md.xsd** and **common.xsd** are in the correct folder. If you are editing the XML in the game folder directly, all is well and the files are loaded from the libraries folder. However, if you are editing in a separate folder, copy those XSD files from the libraries folder directly into the folder where your XML files are located.

> ⚠ Even if your script is free of XSD errors, that does not mean that the script syntax is correct. For example, there are XML elements that require at least one of multiple attributes, but this requirement cannot be reflected in a schema (apart from documentation text). Please notice the XSD documentation of the elements and attributes, e.g. displayed via tooltips in Visual Studio / Visual Web Developer. Please also note additional requirements for MD cue attributes in this guide (see Conditions).
>
> To check for errors, please pay attention to in-game error messages that are produced while your script is imported, and run-time errors while the script runs. The XSD files can help you a lot, but you should not rely on the absence of XSD errors.

Script debug output

The game can print error messages and, when enabled, also general messages. Error messages can originate from the scripting system, but also from other game sub-systems. They can be viewed in the in-game DebugLog.

To collect all messages in a file, start the game with the following parameters on the command line:

```
-logfile debuglog.txt
```

All messages, including enabled non-error messages, will be written into the log file. You can find it in your personal folder, where your save folder is located. To enable scripting-specific debug messages, add the following to the command line:

```
-debug scripts
```

Other debug filters other than "scripts" can be enabled by repeating the -debug command for each filter name, but that is rarely needed for scripting.

The script action <debug_text> can be used to print debug messages from within a script.

MD script structure

In this section we will look at how to start the whole process by creating a new MD mission file and the basic steps in producing mission content with XML code. There will be a description of the key elements of the mission file.

The XML root node of an MD file is called "mdscript" and looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mdscript name="ScriptName" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLoca
tion="md.xsd">
```

"ScriptName" is the name used for this script regardless of the file name. It **has to start with an upper case letter and must be unique** among all MD script names. It also should not contain spaces, so other MD scripts can use it as an identifier to access this script's contents easily.

The only allowed sub-node of <mdscript> is <cues>, which can only contain <cue> sub-nodes:

```
<?xml version="1.0" encoding="utf-8"?>
<mdscript name="ScriptName" ...>
  <cues>
    <cue name="RootCue1"> [...]
    </cue>
    <cue name="RootCue2"> [...]
    </cue>
```

```
  </cues>
</mdscript>
```

Cues

Cues are the main ingredient of an MD script. A cue consists of a set of **conditions** and a set of **actions**. When the conditions are met, the cue is activated and the actions are performed. A cue can have child cues, or **sub-cues**: A sub-cue exists only when its parent cue has become active, so the activation of the parent cue initiates the condition checks of its child cues.

A cue can have the following states:

- **Disabled**: The parent cue has not become active yet, so this cue is basically non-existing.
- **Waiting**: Either this is a root cue, or the parent has become active. The cue is checking its conditions and will become active when they are met.
- **Active**: The cue is about to perform the actions. Child cues have entered the waiting state.
- **Complete**: The cue has finished performing its actions.
- **Cancelled**: The cue has been cancelled. This state cannot normally be reached but only if a cue actively cancels itself or another cue. No condition checks or actions are performed in this cue or any sub-(sub-)cue.

There can be a delay between the activation and performing the actions if the <delay> tag is used. In this case, sub-cues will be enter the waiting state before the parent's actions are performed.

This is how a cue node looks like:

```
<cue name="CueName">
  <conditions> [...]
  </conditions>
  <delay exact="5s" />
  <actions> [...]
  </actions>
  <cues> [...]
  </cues>
</cue>
```

The rules for naming cues is the same for MD script names: The name **starts with an upper case letter**, and has to be **unique within this file**. So it is actually possible to use the same cue name in different scripts, which is different from the MD in X3.

Conditions

The <conditions> node can contain one or multiple conditions, all of which must be met to activate the cue. If the node is missing, the cue will become active unconditionally. The conditions are checked in sequence, and if a check fails, the following conditions are ignored. There are two types of conditions: Events and non-event conditions.

Non-event conditions are checked either once or repeatedly in a fixed interval. They may be based on simple values or ranges, such as a particular in-game time having been reached or the player having a certain amount of money. They may also be based on more complex player information, such as what ships they own, whether the player is in a particular area or near a particular object.

Event conditions are triggered when the corresponding event happens, such as the event that a particular object has been targeted, attacked or destroyed. All event nodes have the prefix “event_” so you can easily determine a condition type. After an event condition you can specify one or more non-event conditions, which will be checked additionally whenever the event happens. If a condition uses an event, it must be in the first sub-node of the <conditions> node. It is even possible to define multiple alternative events that should activate the cue. The first sub-node should be <check_any> in this case, so only one of its sub-conditions has to be met.

Example for an event condition:

```
<conditions>
  <event_object_destroyed object="$target"/>
</conditions>
```

Example for an event condition with an additional (non-event) check:

```
<conditions>
  <event_player_killed_object/>
```

```
  <check_value value="event.param.isclass.turret"/>
</conditions>
```

Example for an event condition with two alternative events and a common additional check:

```
<conditions>
  <check_any>
    <event_cue_completed cue="Cue1"/>
    <check_all>
      <event_player_killed_object/>
      <check_value value="event.param.isclass.turret"/>
    </check_all>
  </check_any>
  <check_age min="$starttime"/>
</conditions>
```

For more information about expressions and event parameters, see below.

<check_all> and **<check_any>** can be used with non-event conditions as well, but if <check_any> is the first node of an event condition, all its sub-nodes have to define events. In case of <check_all>, only its first node must be an event (or yet another <check_any>), to make sure that exactly one event is required to activate the cue.

If a cue has a <conditions> node without any event, it must have one of the attributes ***onfail*** or ***checkinterval***.

- Use *onfail* if the conditions should be checked only once. The possible attribute values are "*cancel*" and "*complete*". If the conditions are met, the cue will activate and perform the cue actions. Otherwise it's a failure and the cue will be cancelled or completed, based on the onfail attribute. Typically *onfail="cancel"* is used to prevent any further action. *onfail="complete"* can be used to continue with the sub-cues even in case of failure (but skipping the current cue actions).
- With *checkinterval*, you can specify a constant time interval between condition checks. The conditions will be checked regularly forever until they are met, unless the cue's state is changed explicitly by an external event.

Additionally, you can use the attribute **checktime** to set the time of the first condition check (also possible in combination with *onfail*). The *checkti me* can be an expression with variables and is evaluated when the cue is enabled (when the condition checks would normally start – for root cues that happens at game start, otherwise after the parent cue becomes active).

Examples:

Check conditions every 5 seconds, but start checking only 1 hour after game start.

```
<cue name="Foo" checktime="1h" checkinterval="5s">
  <conditions>
  [...]
</cue>
```

Check conditions 3 seconds after the cue is enabled, and cancel the cue in case of failure.

```
<cue name="Foo" checktime="player.age + 3s" onfail="cancel">
  <conditions>
  [...]
</cue>
```

The attributes *onfail*, *checkinterval*, *checktime* are not allowed for cues with event conditions.

Reminder: When using an XSD-capable editor, it's a great help, but you cannot rely on that alone to verify correctness. Please also check the documentation and look for errors in the game debug output. Concretely, the schema cannot tell whether the above cue attributes are used correctly.

Actions

The <actions> node contains the actions that are performed one after another, without any delay inbetween. You can enforce a delay after activation of the cue and actual action performance, using a <delay> node right before the <actions>:

```
<delay min="10s" max="30s"/>
```

Note that during the delay the cue is already in the active state, and the sub-cues have been enabled! If you want to make sure that a sub-cue only becomes active after this cue is complete, there is a useful event condition for that:

```
<event_cue_completed cue="parent"/>
```

<actions> is optional. Leaving it out may be useful if you only want to enable sub-cues after the cue's condition check. The state transition from active to complete will still take the <delay> node into account.

Note that the MD script language is not designed as a programming language. The actions are performed in sequence, although they can be nested to form more complex structures. Loops and conditionals exist to some extent, but not necessarily in the sense that a programmer might expect. Analogously to <check_all> and <check_any>, you can use **<do_all>** to perform all the contained sub-node actions, and **<do_any>** to perform only one of them. <do_all> is particularly useful when nested in a <do_any>.

Example, which selects one of the three texts randomly:

```
<actions>
 <do_any>
   <debug_text text="'Hello world'"/>
   <debug_text text="'Welcome to the MD'"/>
   <debug_text text="'And now for something completely different'"/>
 </do_any>
<actions>
```

⚠ Messages printed with <debug_text> are usually only visible when the "scripts" debug filter is enabled, see Script debug output.

Each child action in a <do_any> node can have a ***weight*** attribute, which can be used to control the random selection of an action node. The default weight of a child node is 1.

Also available is **<do_if>**, which completes the enclosed action(s) only if one provided value is non-null or matches another. Directly after a <do_if> node, you can add one or more **<do_elseif>** nodes to perform additional checks only in case the previous conditions were not met. The node **<do_else>** can be used directly after a <do_if> or a <do_elseif>. It is executed only if none of the conditions are met.

<do_while> also exists, but should be used carefully, since it is the only action that could cause an infinite loop, which freezes the game without any chance of recovery.

Every action can have a ***chance*** attribute, if you only want it to be performed with that chance, given as percentage. Otherwise it will simply be skipped. If chance is used on a conditional action such as <do_if>, the script will behave as if the condition check failed.

Libraries

Libraries are cues which are not created directly but only serve as templates for other cues. This allows for modularisation, so you can re-use library cues in many different missions.

⚠ The syntax of libraries is considerably different from the syntax in the MD of X3TC.

Library cues are written like normal cues, they are also defined in a <cues> node, just with the difference that the XML tag is called library instead of cue:

```
<library name="LibFoo" checktime="1h" checkinterval="5s">
  <conditions>
  [...]
</library>
```

Although it is called library, it's basically just a cue that doesn't do anything. You can mix cues and libraries as you want, as root cues or sub-cues - the location within the file is unimportant. All that counts is the library name, which has to be unique within the MD script, like all other cue names.

To use a library, use the attribute ref:

```
<cue name="Foo" ref="LibFoo"/>
```

This will create a cue with the name Foo that behaves just like the library cue LibFoo. In this example, LibFoo has to be a library in the same MD script file. To use a library LibFoo from another script, you have to qualify it with the script name, using the **md** prefix:

```
<cue name="Foo" ref="md.ScriptName.LibFoo"/>
```

When the ref attribute is provided, all other attributes (except for name) will be ignored and taken from the library cue instead. (By default a library creates its own namespace, as if namespace="static" were specified. See the section about namespaces.)

Also all sub-cues of the library will be created as sub-cues of the cue that uses it. They are defined in the library as <cue>, not as <library>. (Although you can define a library as a sub-cue of another library, the location in the file does not matter, as already stated above.) It is even possible to reference other libraries in sub-cues of a library!

In contrast to X3TC, a cue that references a library also has its own name (Foo in the example above), so other cues can access it in expressions by that name. Sub-cues of Foo cannot be accessed by their name though. Within the library itself, expressions can use all names of cues that belong to the library (the <library> and all sub-cues). They will be translated properly when the library is referenced. Examples:

```
<cue name="Foo" ref="LibFoo"/>
<cue name="Bar" ref="LibFoo"/>

<library name="LibFoo">
  <actions>
    <cancel_cue cue="this"/>             <!-- Cancels the cue referencing LibFoo -->
    <cancel_cue cue="LibFoo"/>           <!-- Cancels the cue referencing LibFoo -->
    <cancel_cue cue="Foo"/>              <!-- Error, Foo not found in library -->
    <cancel_cue cue="Baz"/>              <!-- Cancels Baz in the referencing cue -->
    <cancel_cue cue="md.Script.Foo"/>    <!-- Cancels Foo -->
    <cancel_cue cue="md.Script.LibFoo"/> <!-- Error, trying to cancel library -->
    <cancel_cue cue="md.Script.Baz"/>    <!-- Error, trying to cancel library sub-cue -->
  </actions>
  <cues>
    <cue name="Baz"> [...] <!-- Sub-cue is created in all cues referencing LibFoo -->
  </cues>
</library>
```

These examples are definitely not examples of good scripting style.

So when writing the library, you don't have to worry about name confusion, just use the names of cues in your library and it will work as expected when the library is used. Names of cues that do not belong to the library will not be available in expressions (see Foo in the example above), however, names of other libraries in the file are available when referencing them in the ref attribute.

Library Parameters

A library can be parametrised, so that it can be adapted to the needs of a missions that uses it. You can define required and/or optional parameters for a library, and it will be validated at load time that the user of the library has provided all required parameters.

Parameters are defined like this:

```
<library name="Lib" onfail="cancel">
  <params>
    <param name="foo"/>
    <param name="bar" default="42"/>
    <param name="baz" default="player.age"/>
  </params>
  [...]
</library>
```

If a default value is supplied, the parameter is regarded as optional, otherwise it's required. When providing the actual parameters in a referencing cue, note that there is no <params> node:

```
<cue name="Foo" ref="Lib">
 <param name="foo" value="race.argon"/>
 <param name="bar" value="0"/>
</cue>
```

The values (including default values) can be variable expressions and will be evaluated when the cue is enabled, i.e. when it starts checking the conditions. They will be available to the cue as variables, using the parameter name with a '$' prefix. In the example above, the variables $foo, $bar, and $baz would be created.

```
<library name="Lib">
  <params>
    <param name="foo"/>
  </params>
  <actions>
    <debug_text text="$foo"/>
  </actions>
</library>
```

If your library is supposed to provide a result to the library user, it is recommended to store a predefined variable in the library cue with a standardised name, e.g. $result. The user will be able to read it via CueName.$result. This variable does not have to be defined as a parameter but should be documented in the library.

Instantiation

One of the possible cue attributes is ***instantiate***. If you set it to true, this changes what happens when a cue's conditions are met. Normally, if a cue is not instantiated, the cue's actions are run (taking a delay node into account) and the cue is marked as completed. But with *instantiate, a* **co py of the cue** (and all its sub-cues) is made when the conditions are met, and it is this copy in which the actions are performed and it is the copy whose status is set to complete when they are finished - this means that the original cue (the so-called **static cue**) remains in the *waiting* state, and if the conditions are met again then the whole thing happens all over again.

An instantiating cue should only be used with conditions that are only going to be met once (or a fairly limited number of times), or with conditions that include an event condition. Instantiation should not be used in a cue which, say, just depends on the game time being greater than a specific value as this will result in a copy of the cue being made after each check interval, which could increase memory usage a lot. The most common use of an instantiated cue is in responding to events such as the player ship changing sector, to react every time that event happens.

Instances that are created via *instantiate* are called **instantiated cues**. But sub-cues of instances are also instances (**sub-instances**) - they are created when they enter the waiting state. An instance is removed again (thereby freeing its memory) when it is complete or cancelled, and when all its instance sub-cues have been removed before. The simplest case is an instantiating cue with no sub-cues: The instance is created, the actions are performed, and the instance is removed immediately on completion. A pitfall could be an instance with a sub-cue that is forever in the waiting state (e.g. waiting for an event from an already destroyed object). It can never be removed, so you should clean up such a cue yourself, e.g. by cancelling it explicitly.

Cleaning up instances explicitly

Cancelling a cue with **<cancel_cue>** also cancels all its sub-cues, and cancelling a static cue stops it from instantiating more cues - but it does not cancel its instances. Resetting a cue with **<reset_cue>** resets both sub-cues and instantiated cues, but has the (desired) side effect that condition checks will start again if the parent cue's state allows it. Even a sub-instance that has been reset can return to the *waiting* state. Resetting an instantiated cue will stop it forever, because it is not supposed to be in the *waiting* state (only its static cue is). Resetting will also induce the clean-up reliably, but keep in mind that this is not the case for instance sub-cues.

<cancel_cue> and <reset_cue> only take effect after all remaining actions of the current cue are performed. So you can even safely cancel the cue that you are currently in (keyword "**this**") or any ancestor cue, and still perform more actions afterwards.

Access to instances

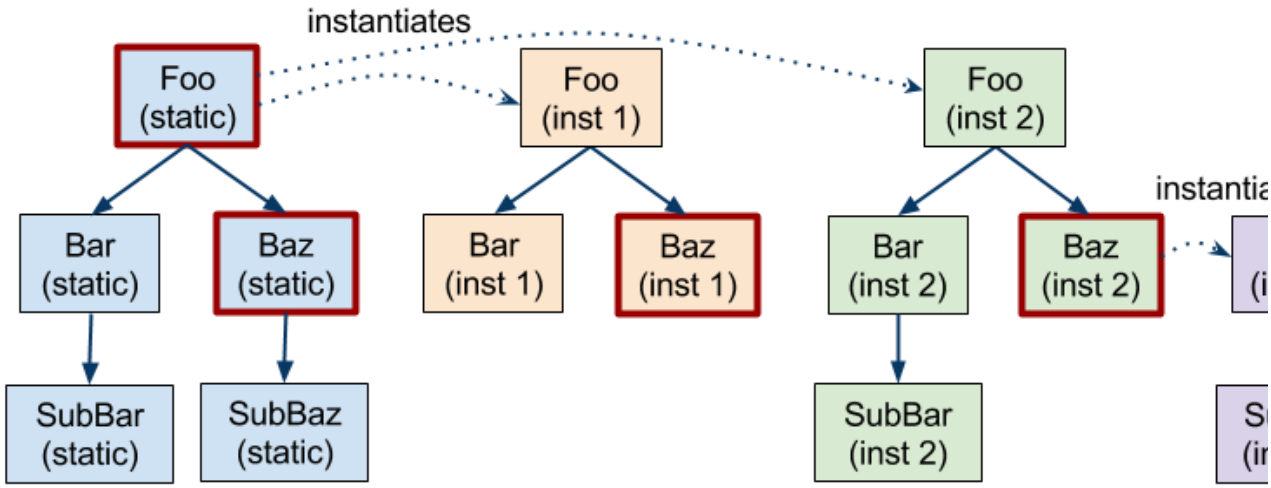
This sub-section requires basic knowledge of script expressions.

In case of instances with sub-instances, you will often want to access a related instance from the current one. Like in the non-instance case, you can simply write the cue name in an expression to reference that cue. However, you should be aware of the pitfalls that are accompanied by this.

When you use a cue name from the same script in an expression, it will always be resolved to some cue - usually a static cue, even if it is still in the disabled state, but it can also be an instance, if it is "related" to the current one.

Related means that this cue and the referenced cue have a common ancestor instance, and the referenced cue is a direct (non-instantiated) descendant of that common ancestor.

Example chart:



This chart represents a script of 5 cues: Foo, Bar, SubBar, Baz and SubBaz. Continuous arrows denote parent-child relationship. Foo and Baz are instantiating cues (highlighted with red border). The static cues always exist, although static children of instantiating cues can never become active. Instances only exist as long as they are needed.

Example situations:

- In the static tree: Cue names in expressions are always resolved to the static cues.
- In the inst-2 tree: “SubBar” in an expression will be resolved to SubBar (inst 2).
- In the inst-1 tree: “SubBar” in an expression will be resolved to SubBar (static) (!) because the SubBar child of Bar (inst 1) does not exist yet, or not any more.
- In the inst-2a tree: “SubBaz” in an expression will be resolved to SubBaz (inst 2a)
- In the inst-2a tree: “Bar” in an expression will be resolved to Bar (inst 2) because Foo (inst 2) is a common ancestor.
- In the inst-2 tree: “SubBaz” in an expression will be resolved to SubBaz (static) (!) because SubBaz (inst 2a) is **not** a direct descendant of the common ancestor Foo (inst 2), instead Baz (inst 2a) has been instantiated.

In expressions, you can use the cue property **static** to access the static cue that instantiated a cue. This does not work for sub-cues of other cues, and the result is not necessarily a real static cue! In the example above, it would only work for cues with a dotted arrow pointing at them, and is resolved to the source of the arrow. In other cases the result is null.

To get the real static cue that always exists and serves as template for instances, use the property **staticbase**. This works for all cues, even for the static cues themselves.

In general, to access ancestors of the current cue, you can also use the keyword **parent**, also recursively as properties of other cues (such as **parent.parent.parent).**

You can store cue references in variables. But when storing an instance cue in a variable, and later accessing that variable, be aware that the instance may not exist any more. Use the property **exists** to check if an instance is still alive. (In contrast, non-instance cues always exist, but may be in the *disabled* or *cancelled* state.)

Pitfalls

Some additional common pitfalls with respect to instantiation are listed here. There may be more.

- **Conditions with results:** If the instantiating cue has conditions with results, those results are stored in variables - but in the variables of the static cue, not of the instance! So in the <actions> you have to access the variables via the **static** keyword:
  ```
  <debug_text text="static.$foo"/>
  ```
  It may even be necessary to copy the variables over to the instance because the static variables can be overwritten by the next condition check:
  ```
  <set_value name="$foo" exact="static.$foo"/>
  ```
- **Resetting completed/cancelled instances:** As explained above, sub-instances are only created when needed (when going to the *waiting* state) and are destroyed when they are not needed any more (when they are completed or cancelled, including all sub-cues). There are cases in which you want to access cues that don’t exist any more - it simply doesn’t work. In some cases you are safe: You can be sure that all your ancestors exist, and instantiating cues won’t be removed until they are cancelled. In some other cases you simply don’t know and have to check if the instance is already (or still) there.
- **Lifetime of instances:** Do not make assumptions about when an instance is removed! Just looking at it in the Debug Manager keeps it alive for the time being. So, sometimes you could still have a completed instance that wouldn’t exist under other circumstances.

Expressions

Most of the attribute values in actions and conditions are interpreted as script expressions and parsed accordingly. An expression is a phrase that can be evaluated to a single value. The simplest expressions are actual numeric values and strings, so called **literals:**

- `0` (integer number)
- `0772` (leading 0 means octal integer number)
- `3.14159` (floating point number)
- `5e12` (float in exponent notation, “times ten to the power of”)
- `0xCAFE` (hexadecimal integer number)

⚠ Since octal numbers are hardly ever used (usually unknowingly), the parser is will produce a warning if an octal number is encountered.

You can write string literals by putting the string in single quotes:

- `'Hello world'`
- `''` (empty string)
- `'String with a line break\n'`

⚠ Since expressions are written in XML attribute values, you have to use the single quotes inside the double quotes for the actual attribute value. To write characters like **< > " &** in an expression string (or anywhere else in an XML attribute value), you’ll have to escape them as **< > " &** respectively. The backslash **** can be used in strings for escape characters like in C/C++. Most important are **\'** for a single quote as part of the string, and **** for the backslash itself.

Numeric data types and suffixes

Numbers can have a suffix that determines their numeric type. There are also numerical data types like “money” or “time” which can only be expressed by using an appropriate unit suffix:

- `5000000000L` (large integer)
- `1f` (floating point number, same as 1.0, just 1 would be an integer)
- `1000Cr` (Money in Credits, converted to 100000 cents automatically)
- `500m` (Length in metres)
- `10s` (Time in seconds)
- `1h` (Time in hours, which is converted to 3600s automatically)

A space between number and suffix is allowed.

Here is the complete list of numeric data types and corresponding unit suffixes:

Data type	Suffix	Examples	Description
null	(none)	null	Converted to non-null data type of value 0 when needed.
integer	i	42	32-bit signed integer. Default for integer literals, so the suffix is not required for them.
largeint	L	0x1fffffffL	Large 64-bit signed integer.
float	f	3.14 0x100f	32-bit float (single precision). Default for floating point literals, so the suffix is not required for them.
largefloat	LF	1.5e300 LF	Large 64-bit floating point number (double precision).
money	ct (default) Cr	200Cr 50ct	Money in Credits or cents, always stored in cents. Do not forget to write Cr when working with Credits.

length	m (default) km	500m 2.3km	Length in metres or kilometres, respectively. A length value is always stored in metres.
angle	rad (default) deg	90deg 3.14159rad	Angle in radians or degrees, respectively. An angle value is always stored in radians.
hitpoints	hp	100hp	Hit points
time	ms s (default) min h	800ms 1.5s 10min 24h	Time in milliseconds, seconds, minutes, or hours, respectively. A time value is always stored in seconds.

⚠ All unit data types are floating point types, except for money, which is an integer data type.

Operators

You can build expressions by combining sub-expressions with operators. For Boolean operations, expressions are considered “false” if they are equal to zero, “true” otherwise. The following operators, delimiters, and constants are supported:

Operator / Delimiter / Constant	Type	Example	Result of example	Description
null	constant	`null + 1`	`1`	Null value, see above
false	constant	`1 == 0`	`false`	Integer value 0, useful in Boolean expressions
true	constant	`null == 0`	`true`	Integer value 1, useful in Boolean expressions
pi	constant	`2 * pi`	`6.2831853rad`	as an angle (same as 180deg)
()	delimiter	`(2 + 4) * (6 + 1)`	`42`	Parentheses for arithmetic grouping
[]	delimiter	`[1, 2, 2+1, 'string']`	`[1, 2, 3, 'string']`	List of values
table[]	delimiter	`table[$foo='bar', {1+1}=40+2]`	`table[$foo='bar', {2}=42]`	Table of values
{}	delimiter	`{101, 3}`	`'Some text'`	Text lookup (page ID and text ID) from TextDB (Note: Braces are also used for property lookups)
+	unary	`+21 * (+2)`	`42`	Denotes positive number (no effect)
-	unary	`-(21 * -2)`	`42`	Negates the following number
not	unary	`not (21 == 42)`	`true`	Yields true if the following expression is false (equal to zero), false otherwise

typeof	unary	`typeof null`
		`typeof 0`
		`typeof 'Hello world'`
			`datatype.null`
			`datatype.integer`
			`datatype.string`
				Yields the data type of the following sub-expression

sin	unary	`sin(30deg)` `sin(pi)`	`0.5` `1.0`	Sine (function-style, parentheses required)

cos	unary	`cos(60deg)` `cos(pi)`	`0.5` `0.0`	Cosine (function-style, parentheses required)
sqrt	unary	`sqrt(2)`	`1.414213LF`	Square root (function-style, parentheses required)
exp	unary	`exp(1)`	`2.71828LF`	Exponential function (function-style, parentheses required)
log	unary	`log(8) / log(2)`	`3.0LF`	Natural logarithm (function-style, parentheses required)
^	binary	`10 ^ 3`	`1000.0LF`	Power
*	binary	`21 * 2`	`42`	Multiplication
/	binary	`42 / 10` `42.0 / 10.0`	`4` `4.2`	Division
%	binary	`42 % 10`	`2`	Modulus (remainder of integer division)
+	binary	`1 + 1` `'Hello' + ' world'`	`2` `'Hello world'`	Addition String concatenation
-	binary	`1 - 1`	`0`	Subtraction
lt < (<)	binary	`1 lt 3` `1 &lt; 3`	`true`	Less than
le <=	binary	`1 le 3` `1 &lt;= 3`	`true`	Less than or equal to
gt > (>)	binary	`1 gt 3` `1 &gt; 3`	`false`	Greater than
ge >=	binary	`1 ge 3` `1 &gt;= 3`	`false`	Greater than or equal to
==	binary	`1 + 1 == 2.0`	`true`	Equal to
!=	binary	`1 + 1 != 2.0`	`false`	Not equal to
and	binary	`true and false`	`false`	Logical AND (short-circuit semantics)
or	binary	`true or false`	`true`	Logical OR (short-circuit semantics)
if ... then ... if ... then ... else ...	ternary	`if 1 == 2 then 'F'` `if 1 == 2 then 'F' else 'T'`	`null` `'T'`	Conditional operator ("inline if")

Operator precedence rules

You can group sub-expressions using parentheses, but if you don't, the following order of operations is applied, so that 5-1+2*3 == 10 as you would expect. The order is the same as in the table above, but there are operators with the same precedence - these are applied from left to right.

- Unary operators: +, -, not, typeof, function-style operators (highest precedence)
- Power operator: ^
- Multiplicative: *, /, %
- Additive: +, -
- Comparison: lt, le, gt, ge
- Equality: ==, !=
- and
- or
- if/then/else (lowest precedence)

Type conversion

When a binary arithmetic operator is used on numbers of different types, they will be converted to a suitable output type. The resulting type depends on whether a unit data type is involved (types that are not plain integers or floats). The following cases may occur:

- Null and something else: The null value will be interpreted as “0” of the other type.
- Two non-unit integers: The result will be an integer of the largest involved type.
- Two non-unit numbers, not all integers: The result will be the largest involved float type.
- Non-unit and unit: The result will be the unit type.
- Two different units: The types are incompatible. This is an error, the result is undefined.

For multiplication and division, this may not be intuitive in all cases: Dividing a length by another length results in a length - so if you want to have a simple float as a result, you will have to convert it manually.

There is a way to convert a number into a different type manually: You append the corresponding suffix to a sub-expression in parentheses, like this:

- `(1 + 1)f 2f 2.0`
- `(1h) m / (180deg) i (3600s) m / (3.14rad) i 3600m / 3 1200m`

When converting to a non-default unit type, this means you interpret the number as in the given units: “`(1km + 500m)h`” means that you interpret 1500m as 1500 hours, so the resulting value will be 1500x3600 seconds. (As stated above, the default unit for a length is metres.)

The division operation will be an integer division (rounding towards zero) if both operands are integers (see the example in the table above). So if you want to get a floating point result, you have to make sure that at least one of the operands is a floating point type.

Every data type can be combined with a string with the + operator, and will be converted to a string representation. That way you can also concatenate strings and numbers:

- `'One plus one is equal to ' + (1+1) + '.' 'One plus one is equal to 2.'`
- `'One plus one is not equal to ' + 1 + 1 + '.' 'One plus one is not equal to 11.'`

As you can see, operators of the same precedence (+ in this case) are always evaluated from left to right.

Boolean operators

Some additional notes on Boolean operators (such as and, or, not, ==):

- Of course a Boolean operation always results in true or false (integer 1 or 0).
- Values of any type can be used as Boolean operands, e.g. for “and”. They will be interpreted as “true” if they are **non-zero** or **non-numeric**.
- != and == can be used with any data types, even non-numeric ones. When comparing two numeric values, they are converted using the rules above. Values of non-numeric types are never equal to null, or to any other numbers.
- “and” and “or” use short-circuit semantics: The right side of the operation can be skipped if the left side already determines the outcome of the operation
  - Example: `false and $foo false` (the value of $foo is not checked at all)
- Unlike != and ==, the comparison operators <, <=, >, >= are only supported **for numeric values**, **difficulty levels**, and **attention levels**. Comparing other non-numeric values will result in an error and an undefined result.
- <, <=, >, >= cannot be used in XML directly, so lt, le, gt, ge are provided as alternatives. In some cases you won’t have to use them, though - using range checks with additional XML attributes can be more readable.

Strings and formatting

You can concatenate string literals using the + operator, but there is also a printf-like formatting syntax, which is easier to use than concatenating lots of small pieces:

- `'The %1 %2 %3 jumps over the %5 %4'.['quick', 'brown', 'fox', 'dog', 'lazy']`
- `'%1 + %2 = %3'.[$a, $b, $a + $b]`

See also the section about value properties.

Instead of ‘%1 %2 %3’, you can also use ‘%s %s %s’, which is also compatible with Lua string formatting in the UI system. However, this should only be used if you are sure that the order is the same in all supported languages. If you want to make translators aware that they can change the order of parameters, you should prefer '%1 %2 %3'.

To get a percent character in the result string, use '%%' in the format string.

If you need a more sophisticated method for text substitution, try **<substitute_text>**. See the XML schema documentation for this script action.

[New as of X Rebirth 4.0]

With the formatting syntax above, it is even possible to control how the parameter is formatted, using modifiers between "%" and the parameter specifier ("s" or the parameter number):

- `'%,s'.[12345678] '12,345,678'` (the "," modifier shows a number with thousands separators, correctly localised)
- `'%.3s'.[123.4] '123.400'` (show 3 fractional digits, rounding half away from zero - decimal point correctly localised)
- `'%,.1s'.[12345.67]' '12,345.7'` (combination of the above)

Additional remarks:

- The "," and "." formatting modifiers only apply to numbers. They are ignored if used on values of other types.
- If "," is used without "." then any fractional digits are discarded.
- "." must be followed by a single digit (0-9). In case of ".0" any fractional digits are discarded (rounding towards zero, not half away from zero).

There are also special methods to format money values and time values using the "formatted" property.

Lists

Another example for a non-numeric value is a list: It is an ordered collection of other arbitrary values (called array or vector in other languages). It can be constructed within an expression using the [] syntax. It may also be generated by special actions and conditions, and there are actions that can insert or remove values.

A list can contain values of arbitrary data types, even mixed in the same list - so a list can actually contain other lists. However, some of the things that you can do with lists require that all contained elements are of a certain type. The contents of a list can be accessed via properties, see the section about value properties. Lists can be empty, these are written as “[ ]”.

When accessing a list's elements, the numbering is **1-based**, so the first element has number 1. This is intuitive but different from 0-based numbering in most programming languages.

Lists are stored in variables as references, so multiple variables can refer to the same **shared list**: If you change a shared list through a variable, e.g. by changing the value of an element, you change it as well for all other variables. However, the operators == and != can also be used on two distinct lists to compare their elements.

When using <remove_from_list/>, be aware that all elements are checked and potentially removed during the action. Do not provide this action with a index lookup of that list as it may become out of bounds.

Bad usage attempting to remove the last element of the list: <remove_from_list name="$List" exact="$List.{$List.count}"/>

If you know the index, simply use <remove_value/> e.g. <remove_value name="$List.{$List.count}"/>

Tables

Tables are associative arrays - they are like lists, but you can assign values to (almost) arbitrary keys, not just to index numbers. A table is constructed within an expression using the table[] syntax. See the section about value properties for how to access the contents of a table. Creating and removing entries works similarly to lists, but instead of inserting, you simply assign a value to a table key. If the key does not exist yet, it will be created.

Almost all values are allowed as table keys, but there are a few exceptions:

- Strings must start with '$', like variables
- null cannot be used as table key (but the number 0 is valid)
- Lists, tables, groups and buildplans cannot be used as table keys

These restrictions only apply to the keys, there are no restrictions for values that you assign to them. For example:

- `table[]` creates an empty table
- `table[{0} = null]` creates a table that maps the number 0 to null
- `table[{'$foo'} = 'bar']` a table that maps the string '$foo' to the string 'bar'
- `table[$foo = 'bar']` exactly the same, just a shorter notation for string keys
- `table[foo = 'bar']` error, 'foo' does not start with a '$'
- `table[{1} = [], {2} = table[]]` a table that maps 1 to an empty list and 2 to an empty table

Just like lists, tables are stored as references, so it's possible that multiple variables reference the same table (see above).

Value properties

Properties are a crucial concept in script expressions. In the previous sections you have seen mostly constant expressions, which are already evaluated when they are parsed at game start. For reading and writing variables and evaluating the game's state, properties are used.

Numbers don't have any properties. Lists, for example, have quite a few of them: You can access the number of elements; and each element is also a property of the list. A ship can have properties like its name, the ship class, its position etc.

You can imagine properties as key/value pairs in an associative mapping: You pass the key, and you get the value as result. For example, the list [42, null, 'text'] has the following mapping:

- 1 42
- 2 null
- 3 'text'
- 'count' 3

As you can see, a property key can be a number or a string. Actually there is no restriction regarding the data type of the key.

You can look up a property by appending a dot and the key in curly braces:

- `[100, 200, 300, 400].{1}` 100 (reading the first element)
- `[100, 200, ['Hello ', 'world']].{3}.{2}` 'world' (second element of the inner list, which is the third element of the outer list)
- `[].{'count'}` 0
- `table[{21} = 42].{21}` 42

In most cases the property key is a fixed string, like "name" or "class". You can write this like above:

- `[42].{'count'}`
- `$ship.{'name'}`
- `$ship.{'class'}`
- `table[$foo='bar'].{'$foo'}`

But it is easier just to write the property key without braces, which is equivalent:

- `[0].count`
- `$ship.name`
- `$ship.class`
- `table[$foo='bar'].$foo`

(In this case, $ship is a variable. All variables start with a "$", so they cannot be confused with keywords.)

A list has even more properties:

- '**random**' returns a randomly chosen element (which requires that the list is non-empty)
- '**min**' and '**max**' return the minimum or maximum (all elements have to be numeric)
    - `[1, 6, 8].min` 1
- '**average**' returns the average (but all element types have to be compatible)
    - `[1, 6, 8].average` 5
- '**indexof**' is followed by another property, and the index of the first occurence of that key in the list is returned, or 0 if it's not in the list
    - `[1, 6, 8].indexof.{8}` 3
- '**clone**' creates a shallow copy of the list (i.e. lists that are contained as elements in the list are not copied, only the reference to them)
    - `[1, 6, 8].clone` [1, 6, 8]

A table has different properties:

- '**clone**' creates a shallow copy of the table
- '**keys**' allows you to access data about the table's keys

However, 'keys' alone will not give you a result. 'keys' must be followed by another keyword to retrieve the desired information, for example:

- `$table.keys.list`: Yields a list of all keys in the table (reliably sorted by key if all keys are numeric)
- `$table.keys.sorted`: Yields a list of all keys in the table, sorted by their associated values (which requires that all values are numeric)
- `$table.keys.random`: A randomly chosen key (which requires that the table is non-empty)

The string formatting syntax that you have seen above is also based on the property system. You basically pass a list as property key to a string. Braces around the brackets are not required, so 'foo'.[...] is just a convenient alternative notation for 'foo'.{[...]}.

Lookup tests and suppressing errors

If you look up a property that does not exist, there will be an error, and the result will be null. To test whether a property exists, you can append a question mark "?" to the lookup, which yields true or false:

- `$list.{5}` The fifth element of a list - however, if $list has less than 5 elements (and if it's also not a table with the key 5), there will be an error
- `$list.{5}?` true if $list exists and has the property 5, false otherwise
- `$table.$key?` Analogously, true if $table exists and has the string property '$key'

The question mark can even be applied to variables:

- `$list` The value stored under the name $list, or an error if there is no such variable
- `$list?` true if the variable exists, false otherwise

To look up the value of a property although it may not exist, you can use the at-sign “@” as prefix:

- `@$list.{5}` The result of the $list lookup if $list exists and has the property 5, otherwise null (without error message)
- `@$list` The list if this variable exists, null otherwise
- `@$list.{5}.{1}` The first element of the fifth element of $list, if it exists, null otherwise

As you can see, an error is already prevented if any link in the property chain does not exist. But use the @ prefix with care, since error messages are really helpful for detecting problems in your scripts. The @ prefix only suppresses property-related error messages and does not change any in-game behaviour.

Static lookups

There are a few data types which are basically enumerations: They only consist of a set of named values, e.g. the “class” data type, which is used for the component classes that exist in the game. For all these static enumeration classes there is a lookup value of the same name, from which you can get the named values as properties by their name. So for the type “class”, there is a value “class” that can be used to access the classes.

Here are a few enumeration classes and corresponding example lookup values:

Data type (= value name)	Examples	Description
class	class.ship class.ship_xl class.space class.weapon	Component classes
purpose	purpose.combat purpose.transportation	Purposes
killmethod	killmethod.hitbybullet killmethod.hitbymissile	Ways to die (already used before destruction)
datatype	datatype.float datatype.component datatype.class datatype.datatype	Script value datatypes
profile	profile.flat profile.increasing profile.bell	Probability distribution profile (see random ranges)
cuestate	cuestate.waiting cuestate.active cuestate.complete	Cue states
level	level.easy level.medium level.veryhard	Mission difficulty levels (comparable with each other using lt, gt, etc.)

attention	attention.insector attention.visible attention.adjacentzone	Attention levels (comparable with each other using lt, gt, etc.)
ware	ware.ore ware.silicon	Wares
race	race.argon race.boron	Races
faction	faction.player faction.argongovernment	Factions

With the *typeof* operator you can get the datatype of any expression and compare it with what you expect, for example:

```
typeof $value == datatype.faction
```

However, you should not compare the type to datatype.string because there are strings that have different data types. To check for a string you should use the datatype's property "**isstring**" instead. For example, to check if the variable $value is a string, use the following term:

```
(typeof $value).isstring
```

There is also the datatype “tag” with the lookup name “tag” - however, this is not an enumeration type. Looking up a value by name never fails, you actually create a tag value for a given name if it does not exist. For example, if you have a typo, like “tag.mision” instead of “tag.mission”, there won’t be an error because any name is valid for a tag, and the tag “mision” is created on its first use.

Player properties

You can access many player-related game properties via the keyword “player”:

- player.**name**: The player’s name
- player.**age**: The passed in-game time since game start
- player.**money**: The money in the player’s account
- player.**ship**: The ship the player is currently on (not necessarily the player's ship), or null if the player is on a station
- player.**primaryship**: The player's own ship (but the player is not necessarily on board)
- player.**entity**: The actual player object
- player.**zone**, player.**sector**, player.**cluster**, player.**galaxy**: Location of the player entity
- player.**copilot**: The co-pilot NPC

The game consists of objects of different classes (zones, ships, stations, NPCs). They have the common datatype "component", however, they have different properties, e.g. NPCs have the property "race", but ships don't.

Safe properties

Most properties cause errors if you use them on non-existing objects, such as destroyed ships. There are a few exceptions:

- exists
- isoperational
- iswreck
- isconstruction
- available
- isclass.(...)

These properties will not cause errors when used on “null” or on a destroyed object (which may still be accessible from scripts in some cases), and produce null or false as results, respectively. (The keyword “available” is used for trades, not for objects. Trades can also become invalid.) However, when using such a property on a different data type like a number, there will still be an error.

Money and time formatting

[New as of X Rebirth 4.0]

Numbers don't have any properties, except for money and time: They have a "**formatted**" property, which allows you to get a custom string representation with more advanced options than the generic formatting method for numbers.

- `$money.formatted.{'formatstring'}`
- `$money.formatted.default` (using default format string '%s')
- `$time.formatted.{'formatstring'}`
- `$time.formatted.default` (using default format string '%T')

In scripts, money is stored in cents, not Credits. The formatted representation always shows the value in Credits, including thousands separators.

When formatting the money value, any specifier (such as '%s') in the format string is replaced by the money value, so usually the format string only consists of this one specifier. The following modifiers can be used between '%' and the specifier character, to enable formatting options:

1-9	Truncation	To enable truncation, specify the number of relevant digits that should be displayed. If the money string is too long, it can be truncated and a metric unit prefix (e.g. k = kilo) is appended. (All digits are shown unless truncation is enabled.)
c	Colouring	If truncation is enabled, the metric unit prefixes (e.g. k, M, G) can be coloured when displayed on the screen, using the escape sequence '\033C'.
.	Cents	Usually money values have no cent part, since cents are not used in accounts or trades. However, single ware prices can have a non-zero cent part. (Cents are not displayed if money is truncated)
_	Spaces	An underscore adds trailing spaces to the result string for better right-aligned display in a tabular layout.

By default, these options are disabled.

More available specifiers (in addition to %s):

- %k: Credits (truncated) in kilo format
- %M: Credits (truncated) in Mega format
- %G: Credits (truncated) in Giga format
- %T: Credits (truncated) in Tera format
- %Cr: Localised "Cr" string
- %%: A % sign

Examples:

- `(1234Cr).formatted.{'%s'}'1,234'`
- `(1234Cr).formatted.default'1,234'` (same as {'%s'})
- `(1234Cr).formatted.{'%.s %Cr'}'1,234.00 Cr'`
- `(1234Cr).formatted.{'%1s'}'1 k'` (rounding towards zero)
- `(1234Cr).formatted.{'%cM'}'0 M'`

For documentation of time format strings, see the Lua function ConvertTimeString() in the Lua function overview.

Examples:

- `(151s).formatted.{'%T'} '00:02:31'`
- `(151s).formatted.default '00:02:31'` (same as {'%T'})
- `(151s).formatted.{'%.3T'} '00:02:31.000'`
- `(151s).formatted.{'%h:%M'} '0:02'`

Complete property documentation

The game files include the HTML file scriptproperties.html in the game's root folder, and more scriptproperties.* files in the libraries folder. Open the HTML file in a browser (only Firefox is officially supported, there may be problems with Chrome). This provides you with a complete list of all supported "base keywords" and properties. To filter in this list, you can enter an expression in the text field:

- Enter the beginning of a base keyword
- Enter $ followed by the data type you are looking for (e.g. "$ship"), as if it were a variable
- To see the properties of a base keyword or data type, enter a dot (".")
- After the dot, you can enter a property name
- You can also enter a dot (".") as first character to search globally for a property

The documentation contains some data types that are no real script data types, but which are useful for documentation purposes. For example, ships and stations are both of datatype "component", but have different properties based on their component class.

MD refreshing and patching

When a saved game is loaded, the saved MD state is restored, but also all MD files are reloaded and changes in them are applied to the MD state. This is called “refresh”. It is also possible to refresh the MD at run-time using the command “refreshmd” on the in-game command line. This is a convenient way to update MD scripts while the game is already running.

Details and restrictions

Here are some noteworthy facts about refreshing scripts and cues, and the restrictions:

- MD scripts and cues are identified by their names. So a script can only be refreshed if it has the same script name as before (file name is irrelevant).
- If there are new script files or new cue nodes (i.e. scripts/cues with new names) they are created and added properly. If you remove script files or cue nodes, the corresponding scripts/cues are removed from the game, including instances.
- As a consequence, you CANNOT rename scripts or cues if you want to refresh them. Doing so would remove the old script or cue and add a new one with the new name.
- You CANNOT change a <cue> to a <library> or vice versa.
- You CANNOT add, remove, or change the "ref" attribute of a cue. But it is possible to remove the whole cue. (If all references to a library are removed you can also remove the library itself.)
- You CANNOT change the cue tree structure, i.e. if you move a cue out of its <cues> node, you also have to change its name (see above). Changing the order of cues within the same <cues> node is possible, however, the order of execution is not reliable anyway.
- You CAN change a library and change/add/remove its sub-cues. This automatically updates all cues that use the library.
- You CAN change library parameters (both in libraries and in referencing cues). However, this does not change the variables of a referencing cue if it is already enabled.
- You CAN change conditions without restrictions. You can even change between event and non-event conditions. If a cue has enabled condition checks, they are aborted and restarted (even if there is no change).
- Adding root cues enables their condition checks immediately (if the module attribute allows it).
- Adding sub-cues to active or complete cues enables their condition checks immediately.
- You CAN change/add/remove <actions>, <force>, <delay>, and all attributes without restrictions, except for the "ref" attribute (see above). You can even change the <delay> while the cue is already active and the timer is running.
- Changing instantiate="false" to "true" turns the cue into "waiting" state if it was active or complete before.
- Changing instantiate="true" to "false" removes all instantiated cues and their descendants.

> Be aware that completed instances can be auto-deleted, and so added sub-cues will not become active in such a case.

> When adding a variable in a new MD script version and using that variable in multiple places, be aware that the variable doesn't exist yet in older savegames. You may have to check the existence of the variable before accessing it, or add some patch logic that initiailses the variable after loading the savegame, if necessary.

Patching

Cues can have **<patch>** elements with actions that will be performed when an old savegame is loaded. To control which savegames should be affected, you can add a ***version*** attribute to the <cue> node and a ***sinceversion*** attribute in the patch. When a cue is loaded from a savegame that has an older version than *sinceversion*, the <patch> actions will be performed immediately after loading.

```
<cue [...] version="42">
  <conditions> [...] </conditions>
  <actions> [...] </actions>
  <patch sinceversion="42">
    [patch actions]
  </patch>
</cue>
```

The patch actions are only performed if the cue is in a certain state, “complete” by default. Use the ***state*** attribute to change this requirement. For more information, see the XML schema documentation of the <patch> element.

A sequence of multiple <patch> elements is possible. They will be performed in order of appearance, checking the *sinceversion* and *state* attributes in each case. Patches are also applied to all users of a library and to instances.

The <patch> elements will be ignored when refreshing the MD at run-time. They only affect loaded savegames.

Common attribute groups

There are many commonly used actions and conditions which share groups of attributes. The most important ones are explained here.

Value comparisons

There are many conditions and conditional actions that require a value comparison, for example the condition <check_value>:

```
<check_value value="$ware == ware.silicon and $amount != 0"/>
```

In the value attribute you specify a boolean expression, and if it is true (that is, not equal to zero), the condition is met. This is a special case: This condition and all other nodes that support a value comparison allows you to specify an upper limit, a lower limit, a number range, or a list of allowed values. Examples:

```
<check_value value="FooCue.state" exact="cuestate.complete"/>
<check_value value="$foo.count" min="5"/>
<check_value value="$foo" max="player.age + 1min"/>
<check_value value="player.money" min="300Cr" max="600Cr"/>
<check_value value="$method" list="[killmethod.hitbymissile, killmethod.collected]"/>
<check_value value="$attention" min="attention.visible"/>
```

Values of most enumeration types cannot be compared via *min* or *max* (also not via lt, gt, etc.). The only data types that can be used with *min* and *max* are numbers and the enumeration types *level* and *attention* (see Boolean operators). The *exact* attribute can be used with any type, and is equivalent to using the == operator.

Random ranges

If an action requires a value, e.g. when you set a variable to a value, you can have some randomisation. To specify an exact value, e.g. in <set_value>, you can write this:

```
<set_value name="$race" exact="race.teladi"/>
```

To select a random element from a list, this syntax can be used:

```
<set_value name="$prime" list="[2, 3, 5, 7, 11]"/>
```

To get a random number within a given range, you can use min/max:

```
<set_value name="$foo" min="-20" max="20"/>
<set_value name="$timeout" max="20s"/>
```

min and max have to be compatible number types. Enumeration types are not allowed, not even level and attention. The min attribute is optional and defaults to 0 (of the number type used in max).

You can select one of 5 different probability distribution profiles for the random range, “flat” being the default (all values in the range are equally likely). If you select another profile, e.g. “increasing” to make higher numbers more likely, you also have to specify a scale value (integer) that is greater or equal to 2. Higher scale values result in higher peaks in the distribution profiles (probable values become even more probable).

```
<set_value name="$foo" min="-20" max="20" profile="profile.increasing" scale="4"/>
```

Variables and namespaces

As you have seen above, you can easily access variables by writing their name (including $ prefix) in an expression. Namespaces define in which cue the variables are actually stored (and from which cue they are read).

Creating and removing variables

You can create variables with certain actions and conditions, such as the **<set_value>** action:

```
 <set_value name="$foo" exact="$bar + 1" />
```

<set_value> also exists as a “condition”, which can be useful if you want to pass information about the conditions to the actions, that would otherwise be lost - like in a complex <check_any> event condition, where you want to create a variable only if you are in a certain check branch. (Other pseudo-conditions are <remove_value> and <debug_text>.)

The default operation of <set_value> is “**set**”, but there are more: “**add**”, “**subtract**”, and “**insert**”. *add* and *subtract* change the value of an existing variable, which is created as 0 if it didn’t exist before. If neither *min*, *max* nor *exact* attribute is provided, an exact value of 1 is assumed.

```
<set_value name="$foo" operation="add" />
```

The trick is that <set_value> not only works on variables, but also on list elements and table keys:

```
<set_value name="$list.{1}" exact="42" />
<set_value name="$table.$foo" exact="42" />
```

The operation *insert* is special, and it only works on lists. It inserts the value at the specified position (note that the position beyond the last element is also valid here):

```
<set_value name="$list.{1}" exact="42" operation="insert" />
```

This shifts the positions of all following elements up by one. If min/max/exact are missing, the default value is null for insertions, not 1 like in other cases.

Appending is easier than that. The following actions are equivalent:

```
<set_value name="$list.{$list.count + 1}" exact="42" operation="insert" />
<append_to_list name="$list" exact="42" />
```

Inserting at a position below 1 or above $list.count + 1 is not possible.

To remove variables or list/table entries, use <remove_value>:

```
<remove_value name="$foo" />
<remove_value name="$list.{1}" />
<remove_value name="$table.$foo" />
```

Removing an entry from a list shifts all following elements down by one. If you want to clear an entry without removing it from the list, just use <set_value> instead.

Accessing remote variables

You can also read and write variables in other cues by using the variable name as property key:

```
<set_value name="OtherCue.$foo" min="0.0" max="1.0" />
<set_value name="md.OtherScript.YetAnotherCue.$bar" exact="OtherCue.$foo" />
```

Instead of referencing a cue by name, you could also reference it via a keyword or another variable:

```
<set_value name="static.$counter" operation="add" />
<set_value name="parent.$foo" exact="42" />
<set_value name="this.$bar" exact="parent" />
<set_value name="$baz" exact="this.$bar.$foo" />
```

Namespaces

In the examples above, a variable was written to and read from the “this” cue. This can be necessary: the expression “$foo” may be different from the expression “this.$foo”. The reason for that are namespaces.

Consider this case:

```
<cue name="Root">
  <actions>
    <set_value name="$foo" />
```

```
  </actions>
  <cues>
    <cue name="SubCue"> [...]
    </cue>
  </cues>
</cue>
```

When the root cue creates $foo, the variable is stored in the Root cue directly. But SubCue and its descendants will also need access to $foo. Of course they could write “parent.$foo” or “Root.$foo”, but since it’s very common to have a single location for most variables in the whole cue tree, the easy solution is to write just “$foo” - because variable names are looked up in the **namespace cue**, which is the root by default. Also newly created variables end up in the namespace, and not in “this” cue.

You can also use the keyword “**namespace**” in expressions to get the namespace cue.

Defining a cue’s namespace

When writing a cue, you can specify what the namespace of the cue should be, by adding the ***namespace*** attribute. The following values are possible:

- **this**: Use “this” cue as namespace, even for instances: $foo == this.$foo
- **static**: Same as “this”, but when instantiated, use the static cue: $foo == static.$foo
- **default**: The namespace is inherited from the parent cue. The default for root cues and for libraries is the same as “static”.

Although in general the expression “$foo == namespace.$foo” is true, there is one exception: When library parameters are evaluated in the referencing cue, variables are resolved using the parent’s namespace. However, the referencing cue creates a new namespace, so the namespace keyword already points to the library, not to the parent’s namespace. Example:

```
<cue name="LibRef" ref="Lib">
  <param name="Param1" value="$foo" /> <!-- $foo from parent namespace -->
  <param name="Param2" value="namespace.$foo" /> <!-- LibRef.$foo (error) -->
</cue>
```